April 1997

WOL: A Language for Database Transformations and Constraints

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Anthony S. Kosky
University of California

WOL: A Language for Database Transformations and Constraints

Abstract
The need to transform data between heterogeneous databases arises from a number of critical tasks in data management. These tasks are complicated by schema evolution in the underlying databases, and by the presence of non-standard database constraints. We describe a declarative language, WOL, for specifying such transformations, and its implementation in a system called Morphase. WOL is designed to allow transformations between the complex data structures which arise in object-oriented databases as well as in complex relational databases, and to allow for reasoning about the interactions between database transformations and constraints.

Comments
Copyright 1997 IEEE. Reprinted from *Proceedings of the 13th International Conference on Data Engineering*, April 1997, pages 55-65.

WOL: A Language for Database Transformations and Constraints *

Susan B. Davidson
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Email: susan@central.cis.upenn.edu

Anthony S. Kosky
Lawrence Berkeley National Laboratory,
1 Cyclotron Road,
Berkeley, CA 94705.
Email: Anthony_Kosky@lbl.gov

Abstract

The need to transform data between heterogeneous databases arises from a number of critical tasks in data management. These tasks are complicated by schema evolution in the underlying databases, and by the presence of non-standard database constraints. We describe a declarative language, WOL, for specifying such transformations, and its implementation in a system called Morphase. WOL is designed to allow transformations between the complex data structures which arise in object-oriented databases as well as in complex relational databases, and to allow for reasoning about the interactions between database transformations and constraints.

1 Introduction

Problems such as reimplementing legacy systems, adapting databases to schema evolution, integrating heterogeneous databases, and mapping between interfaces and the underlying database all involve some form of transformation of data. Implementing such transformations is a critical task in data management.

In all such data transformations the problem is one of mapping instances of one or more *source* database schemas to an instance of some *target* schema. The schemas involved may be expressed in a variety of different data-models, and implemented using different DBMSs or other kinds of data repositories. Incompatibilities between the sources and target exist at all levels – the choice of data-model and DBMS, the representation of data within a model, the value of an instance – and must be explicitly resolved within the mappings.

Example 1.1: As a simple example, consider the problem of integrating the US Cities-and-States and European-Cities-and-Countries databases shown in Figures 1 and 2. The graphical notation used here is inspired by [2]: the boxes represent *classes* which are finite sets of objects; the arrows represent *attributes*, or functions on classes.

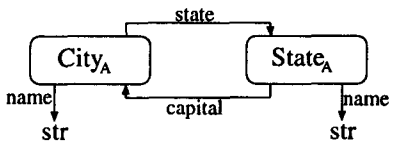

Figure 1: Schema for US Cities and States

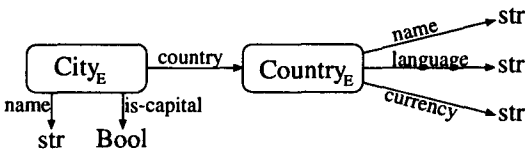

Figure 2: Schema for European Cities and Countries

The first schema has two classes: $City_A$ and $State_A$. The $City_A$ class has two attributes: *name*, representing the name of a city, and *state*, which points to the state to which a city belongs. The $State_A$ class also has two attributes, representing its name and its capital city.

The second schema also has two classes, this time $City_E$ and $Country_E$. The $City_E$ class has attributes representing its name and its country, but in addition has a Boolean-valued attribute *is_capital* which represents whether or not it is the capital city of a country. The $Country_E$ class has attributes representing its name, currency and the language spoken.

Suppose we wanted to combine these two databases into a single database containing information about both US and European cities. A possible schema is shown in Figure 3, where the "plus" node indicates a variant. Here the *City*

*This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, ARPA N00014-94-1-1086 and DOE DE-AC03-76SF00098.

classes from both source databases are mapped to a single class $City_T$ in the target database. The *state* and *country* attributes of the *City* classes are mapped to a single attribute *place* which takes a value that is either a *State* or a *Country*. A more difficult mapping is between the representations of capital cities of European countries. Instead of representing whether a city is a capital or not by means of a Boolean attribute, the *Country* class in our target database has an attribute *capital* which points to the capital city of a country. To resolve this difference in representation a straightforward embedding of data will not be sufficient. Constraints on the source database, ensuring that each *Country* has exactly one *City* for which the *is_capital* attribute is true, are also necessary in order for the transformation to be well defined. ∎

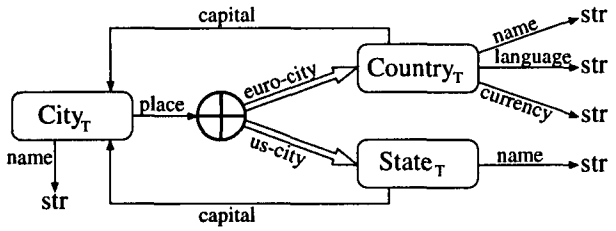

Figure 3: An integrated schema of European and US Cities

This example illustrates a number of complex types: object identities, recursive types and variants. In general, the types of data sources that we are considering are complex object systems, whose types involve arbitrarily deep nesting of records, sets, variants and lists in addition to object identity and the usual base types (Boolean, integer, string, etc). The number of fields in a record or variant may also be extremely large (tens of fields is common), and fields may be optional.

While some commercial solutions exist for transforming data between relational databases implemented using specific DBMSs, or for uploading certain file formats into a relational database, none exist for the variety of data types we are considering. To date, these transformation problems have been attacked by writing special-purpose programs doing explicit data conversions between fixed schemas. This code is typically difficult to understand and reason about, and cannot easily be maintained in the face of schema evolution. It is also difficult to reason about the correctness of the transformation implemented.

Most of the existing work on transformations focuses on the problem of *database integration*. The most common approach taken is to apply a series of small transformations or heuristics to source schemas in order to transform them into the target schema [20, 6, 19, 22]. There are two problems with these approaches. Firstly, the expressibility is inherently limited by the selection of transformations or heuristics supported. For example, none of the systems mentioned would be able to deal with the transformation between the Boolean *is_capital* attribute of Cites and the *capital* attribute of *Country* in the example above. Secondly, these approaches focus on schema manipulation and neglect to describe the effect of the transformations on the actual data, though in general there are many possible interpretations of a particular schema manipulation. For example, if we changed an attribute of a class from being optional to being required, there are a number of ways that such a manipulation can be reflected on the underlying data: we could insert a default value for the attribute wherever it is omitted, or we could simply delete any objects from the class for which the attribute value is missing.

An alternative approach is to use some high-level language to describe transformations as in [3, 9]. In such an approach, the effect of a transformation on the underlying data becomes explicit. We use the term *database transformation*, as opposed to the more common term *schema transformation*, to emphasize this distinction.

A database transformation language should be sufficiently expressive to specify all ways in which data might relate between one or more source databases and a target database, but differs from a database query language in that entire database instances are being manipulated and created. Expressivity must therefore be carefully balanced with efficiency. In particular, an implementation of a transformation should be performed in one pass over the source databases, curtailing the inclusion of expensive operations such as closure operators. Such a language should also be simple and declarative so that it can be easily modified and reasoned about, and should be able to handle the data types found in the formats and databases to be transformed.

The size, number and complexity of schemas that may be involved in a transformation also leads to a need for partiality of rules or statements of a transformation language, and for the ability to reason with constraints. Schemas can be complex, involving many, deeply nested attributes. Values for attributes of an object in a target database may be drawn from many different source database instances. It is therefore very useful to be able to specify the transformation in a step-wise manner in which individual rules do not completely describe a target object.

Constraints can play a part in determining and optimizing transformations, and conversely, transformations can imply constraints on their source and target databases. Further, the constraints that occur when dealing with transformations often fall outside of those supported by most data-models (keys, functional and inclusion dependencies and so on) and may involve multiple databases. It is therefore important

that a transformation language be capable of expressing and interacting with a large class of constraints.

Using these desiderata, we have designed a database transformation language called *WOL* (*Well-founded Object Logic*) for specifying transformations, and developed a system called *Morphase*[1] for implementing transformations specified using *WOL*. *Morphase* has been used in several transformations involving biomedical databases in informatics support of the Philadelphia Human Genome Center for Chromosome 22, and many of the requirements on *WOL* have been drawn from this experience. The *WOL* language has been used independently in part of the VODAK project, in Darmstadt, Germany, for building a data-warehouse, *ReLiBase*, which is used in drug development [1]. In Section 2 we establish the data-model on which *WOL* is based. Section 3 describes *WOL* through a series of examples and illustrates how *WOL* can be used to express a wide variety of database constraints as well as transformations. Section 4 discusses how constraints and transformation clauses interact with each other. Section 5 describes the *Morphase* system which implements *WOL*. Finally, Section 6 summarizes our contributions and describes future work.

2 A Data Model for Database Transformations

To perform transformations between heterogeneous databases, we must first represent the schemas and data of the component databases using some sufficiently expressive common data-model, or *meta-data-model*. In [21] the requirements on such a meta-data-model are examined, and the authors conclude that one which supports complex data-structures (sets, records and variants), object-identity and specialization and generalization relations between object classes is desirable. These conclusions apply equally well to other applications of database transformations.

When transforming recursive data-structures such as those of Figures 1, 2 and 3, it is also necessary to have a notion of *extents* or *classes* in addition to a notion of referencing. These classes represent the finite sets of objects represented in a database. The *WOL* data-model was designed with these requirements in mind, and includes support for object-identities, classes and complex data-structures. The model is similar to that of [4] and is equivalent to the models implemented in various object-oriented databases [5], except for the omission of direct support for inheritance or specializations, which we regard as special kinds of constraints. Constraints are not considered to be part of the model, and are expressed in the *WOL* language as illustrated in the next section. A more detailed definition of the model can be found in [17].

[1] *Morphase* has no relation to the god of slumber, Morpheus, rather it is an enzyme (-ase) for morphing data.

2.1 Schemas and Instances

The types in our model are nested relational types with the additional feature of *class types*. In order to describe a particular database system it is necessary to state what classes are present, and also the types of (the values associated with) the objects of each class. We consider that these two pieces of information constitute a database *schema*. (Other constraints, which in some data-models would also be considered part of a schema, will be considered later).

Assume a finite set $\mathcal{C}$ of *classes* ranged over by $C, C', \ldots$, and for each class C a countable set of *object identities* of class C. The **types** over $\mathcal{C}$, ranged over by $\tau, \ldots$, consist of *base types*, $\underline{b}$, such as *integer* and *string*; *class types* C, where $C \in \mathcal{C}$, representing objects-identities of class C; *set types* $\{\underline{b}\}$ and $\{C\}$ for each base type $\underline{b}$ and class type C, representing sets of base values or objects respectively; *record types* $(a_1 : \tau_1, \ldots, a_k : \tau_k)$, where $a_1, \ldots, a_k$ are taken from some countable set of *attribute labels* $\mathcal{A}$; and *variant types* $\langle\!|a_1 : \tau_1, \ldots, a_k : \tau_k|\!\rangle$. A value of a record type $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ is a tuple with k fields labeled by $a_1, \ldots, a_k$, such that the value of the ith field, labeled by a_i, is of type τ_i. A value of a variant type $\langle\!|a_1 : \tau_1, \ldots, a_k : \tau_k|\!\rangle$ is a pair consisting of a label a_i, where $1 \leq i \leq k$, and a value of type τ_i.

A **schema**, $\mathcal{S}$, consists of a finite set of classes, $\mathcal{C}$, and for each class $C \in \mathcal{C}$ a corresponding type τ^C where τ^C is not a class type.

Example 2.1: The first schema illustrated in example 1.1 has two classes representing *Cities* and *States*, with each city having a *name* and a *state*, and each state having a *name* and a *capital city*. The set of classes for the schema is therefore $\mathcal{C}_A \equiv \{City_A, State_A\}$ and the associated types are

$$\begin{aligned} \tau^{City_A} &\equiv (name : str,\ state : State_A) \\ \tau^{State_A} &\equiv (name : str,\ capital : City_A) \end{aligned}$$

The second schema has classes $\mathcal{C}_E \equiv \{City_E, Country_E\}$ and associated types

$$\begin{aligned} \tau^{City_E} &\equiv (name : str,\ is\_capital : Bool, \\ &\qquad country : Country_E) \\ \tau^{Country_E} &\equiv (name : str,\ language : str, \\ &\qquad currency : str) \end{aligned}$$

■

The values that may occur in a particular database instance depend on the object identities of that instance. Suppose

we have a schema $\mathcal{S}$ with classes $\mathcal{C}$. An **instance**, $\mathcal{I}$, of $\mathcal{S}$ consists of a finite set of object identities, σ^C, for each class $C \in \mathcal{C}$, and a mapping $\mathcal{V}^C$ from σ^C to values of type τ^C, for each $C \in \mathcal{C}$, such that for any object identity $o \in \sigma^C$, the object identities occurring in the value $\mathcal{V}^C(o)$ are contained in the set $\bigcup_{C \in \mathcal{C}} \sigma^C$.

Example 2.2: Continuing with our example database of European cities and countries described in example 2.1, instance of the schema would consist of two sets of object identities, such as

$$\sigma^{City_E} \equiv \{London, Manchester, Paris, Berlin, Bonn\}$$
$$\sigma^{Country_E} \equiv \{UK, FR, GM\}$$

and functions $\mathcal{V}^{City_E}$ on σ^{City_E} and $\mathcal{V}^{State_E}$ on σ^{State_E}, such as

$$\mathcal{V}^{City_E}(London) \equiv (name \mapsto \text{"London"}, country \mapsto UK, is\_capital \mapsto True)$$
$$\mathcal{V}^{City_E}(Manchester) \equiv (name \mapsto \text{"Manchester"}, country \mapsto UK, is\_capital \mapsto False)$$
$$\mathcal{V}^{City_E}(Paris) \equiv (name \mapsto \text{"Paris"}, country \mapsto FR, is\_capital \mapsto True)$$
$$\mathcal{V}^{Country_E}(UK) \equiv (name \mapsto \text{"United Kingdom"}, language \mapsto \text{"English"}, currency \mapsto \text{"sterling"})$$
$$\mathcal{V}^{Country_E}(FR) \equiv (name \mapsto \text{"France"}, language \mapsto \text{"French"}, currency \mapsto \text{"franc"})$$

and so on. ∎

2.2 Surrogate Keys

We must also have some mechanism to create and reference object-identities. Since object identities are not considered to be directly visible and are typically unrelated between databases, some value-based handle on them is necessary. We follow [14] in using surrogate keys for this purpose.

A **key specification**, $\mathcal{K}$, for a schema $\mathcal{S}$, assigns a set of functions $\mathcal{K}^C_{\mathcal{I}}$, $C \in \mathcal{C}$, to each instance $\mathcal{I}$ of $\mathcal{S}$, such that $\mathcal{K}^C_{\mathcal{I}}$ maps σ^C onto values of some type κ^C, where κ^C does not involve any classes.

An instance $\mathcal{I}$ of schema $\mathcal{S}$ is said to *satisfy* a key specification $\mathcal{K}$ on $\mathcal{S}$ iff for each class $C \in \mathcal{C}$ and any $o, o' \in \sigma^C$, if $\mathcal{K}^C_{\mathcal{I}}(o) = \mathcal{K}^C_{\mathcal{I}}(o')$ then $o = o'$.

A **keyed schema** consists of a schema $\mathcal{S}$, and a key specification $\mathcal{K}$ on $\mathcal{S}$. An instance of a keyed schema $(\mathcal{S}, \mathcal{K})$ is an instance $\mathcal{I}$ of $\mathcal{S}$ such that $\mathcal{I}$ satisfies $\mathcal{K}$.

Example 2.3: For the European Cities and Countries schema defined in example 2.1 we might expect each *Country* to be uniquely determined by its name, and each *City* to be uniquely determined by its name and the name of its country (two Countries might both contain Cities with the same name). The functions are defined by

$$\mathcal{K}^{Country_E}_{\mathcal{I}}(x) \equiv x.name$$
$$\mathcal{K}^{City_E}_{\mathcal{I}}(x) \equiv (name = x.name, country\_name = x.name.name)$$

where the notation $x.a$ means if $x \in \sigma^C$ then take the value $\mathcal{V}^C(x)$, which must be of record type, and project out the attribute a. ∎

3 The *WOL* Language

The transformation specification language *WOL* is based on the data-model of the previous section, and can therefore deal with databases involving object-identity and recursive data-structures as well as complex and arbitrarily nested data-structures. A formal definition of *WOL*, its semantics, and the various requirements for a well-defined *WOL* transformation program may be found in [17].

As will be illustrated in section 4, there are important interactions between transformations and the constraints imposed on databases. Since the constraints that are useful when dealing with transformations often fall outside of the simple constraints coupled with traditional data models (such as keys, functional and inclusion dependencies, cardinality constraints and inheritance [10, 12]), *WOL* augments a simple data-model with a general formalism for expressing constraints as well as transformations, making it possible to reason about the interaction between transformations and constraints.

3.1 Formulae and Clauses

A specification of a transformation written in *WOL* consists of a finite set of *clauses*, which are logical statements describing either constraints on the databases being transformed, or part of the relationship between objects in the *source* databases and objects in the *target* database. Each clause has the form

$$head \Longleftarrow body$$

where *head* and *body* are both finite sets of *atomic formulae* or *atoms*. An example of a simple clause for the Cities and States database shown in figure 1 would be

$$X.state = Y \Longleftarrow Y \in State_A, X = Y.capital; \quad (C1)$$

Here the body atoms are $Y \in State_A$ and $X = Y.capital$, and the head atom is $X.state = Y$. Each atom is a basic

logical statement, for example saying that two expressions are equal or one expression occurs within another.

The meaning of a clause is that if all the atoms in the body are true then the atoms in the head are also true. More precisely, a clause is *satisfied* iff, for any instantiation of the variables in the body of the clause which makes all the body atoms true, there is an instantiation of any additional variables in the head of the clause which makes all the head atoms true.

So the clause above says that for any object Y occurring in the class $State_A$, if X is the *capital* city of Y then Y is the *state* of X. This is an example of a *constraint*. We can also use constraints to define the *keys* of a schema that can be used to uniquely identify objects. In our database of Cities, States and Countries, we would like to say that a Country is uniquely determined by its *name*, while a City can be uniquely identified by its *name* and its *country*. This can be expressed by the clauses

$$X = Mk^{City_T}(name = N,\ country = C) \qquad (C2)$$
$$\Leftarrow\ X \in City_T,\ N = X.name,\ C = X.country;$$

$$Y = Mk^{Country_T}(N) \qquad (C3)$$
$$\Leftarrow\ Y \in Country_T,\ N = Y.name;$$

Mk^{City_E} and $Mk^{Country_E}$ are examples of *Skolem functions*, which create new object identities associated uniquely with their arguments. In this case, the *name* of a City and the *country* object identity are used to create an object identity for the City.

WOL can be used to express a wide variety of constraints, including functional and existence dependencies, key constraints, and other kinds of constraints supported by established data-models. It can also express constraints which cannot typically be expressed in the constraint languages of databases. For example, the following constraints expresses that, in our European Cities and Countries database, each country has exactly one capital city.

$$Y \in City_E,\ Y.country = X,\ Y.is\_capital = True \qquad (C4)$$
$$\Leftarrow\ X \in Country_E$$

$$X = Y \Leftarrow\ X \in City_E,\ Y \in City_E, \qquad (C5)$$
$$X.country = Y.country,$$
$$X.is\_capital = True,\ Y.is\_capital = True$$

The first clause states that, for every Country, there is a corresponding City for which the attribute *is_capital* has the value *True*. The second clause states that, for any two Cities belonging to the same Country, if both are capital cities (the *is_capital* attribute has the value *True*) then they are the same City.

Not all syntactically correct *WOL* clauses are meaningful. We require two conditions to hold on a well-formed *WOL* clause, namely that it be *well-typed* and *range-restricted*. A clause is said to be **well-typed** iff we can assign types to all the variables in the clause in such a way that all the atoms of the clause make sense. For example a clause containing the atom $X < Y.population$ (where *population* is an integer valued attribute) and an atom $X \in City_A$ would not be well-typed. For the first atom to make sense X would have to have type *integer*, and for the second it would have to be an object of class $City_A$.

The concept of **range-restriction** is used to ensure that every variable in the clause is bound to some object or value occurring in the database instance in order for the atoms of a clause to be true. This is similar to the idea of *safety* in Datalog clauses. For example the in clause

$$X.population < Y \Leftarrow\ X \in City_A$$

the variable Y is not range restricted.

All the clauses we consider in this paper will be both well-typed and range-restricted.

3.2 Expressing Transformations using *WOL*

In addition to expressing constraints about individual databases, *WOL* clauses can be used to express relationships between the objects of distinct databases. Consider the clause

$$X \in Country_T,\ X.name = E.name, \qquad (T1)$$
$$X.language = E.language,\ X.currency = E.currency$$
$$\Leftarrow\ E \in Country_E;$$

This states that, for every *Country* in our European Cities and Countries database (Figure 2), there is a corresponding *Country* in our target international database (Figure 3) with the same name, language and currency. This is an example of a *transformation clause*, which states how an object or part of an object in the target database arises from various objects in the source and target databases.

A similar clause can be used to describe the relationship between European *City* and *City* in our target database:

$$Y \in City_T,\ Y.name = E.name, \qquad (T2)$$
$$Y.place = ins_{euro\text{-}city}(X)$$
$$\Leftarrow\ E \in City_E,\ X \in Country_T,$$
$$X.name = E.country.name;$$

Note that the body of this clause refers to objects both in the source and the target databases: it says that if there is a City, E, in the European Cities database and a Country, X, in the target database with the same *name* as the *name* of the *country* of E, then there is a City, Y, in the target database with the same *name* as E and with *country* X. ($ins_{euro\text{-}city}$ accesses the *euro_city* choice of the variant).

A final clause is needed to show how to instantiate the *capital* attribute of *City* in our target database:

$$X.capital = Y \qquad (T3)$$
$$\Longleftarrow X \in Country_T,\ Y \in City_T,$$
$$Y.place = ins_{euro\text{-}city}(X),\ E \in City_E,$$
$$E.name = Y.name,\ E.state.name = X.name,$$
$$E.is\_capital = \text{True};$$

Notice that the definition of *Country* in our target database is spread over multiple *WOL* clauses: clause (T1) describes a country's *name*, *language* and *currency* attributes, while clause (T3) describes its *capital* attribute. This is an important feature of *WOL*, and one of the main ways it differs from other Horn-clause logic based query languages such as Datalog or ILOG which require each clause to completely specify a target value. It would be possible to combine clauses (T1), (T3) and (C3), in order to a single clause completely describing how a Country object in the target database arises. However, when many attributes or complex data structures are involved, or a target object is derived from several source objects, such clauses become very complex and difficult to understand. Further if variants are involved, the number of clauses required may be *exponential* in the number of variants involved. Consequently, while conventional logic-based languages might be adequate for expressing queries resulting in simple data structures, in order to write transformations involving complex data structures with many attributes, particularly those involving variants, it is necessary to be able to split up the specification of the transformation into small parts involving partial information about data structures.

A **transformation program** consists of a finite set of transformation clauses and constraints for some source and target database schemas. Given such a transformation program, say **Tr**, a **Tr**-transformation of an instance of the source database would be an instance of the target database such that the two instances satisfy all the clauses in **Tr**.

Since *WOL* clauses represent logical statements, there may be many instances of a target database satisfying a set of clauses for a particular source database. For example the clauses (T1), (T2) and (T3) above would imply that there are objects in our target *Country* class corresponding to each *Country* in our source database, but would not rule out the possibility of having lots of other Countries, not related to any in our source database. When dealing with transformation programs, we are therefore interested in the *unique smallest* transformation of a particular source database.

A transformation program **Tr** is said to be **complete** iff whenever there is a **Tr**-transformation of a particular source database instance, there is a unique smallest such **Tr**-transformation (up to renaming of object identities). In general, if a transformation program is not complete, it is because the programmer has left out some part of the description of the transformation.

3.3 Related Languages

There are a number of database *query* languages based on Horn-clause logic which have some similarity to *WOL*. Most of these languages are variants and extensions of Datalog [?], and are therefore limited in their types to flat relations. For example, ILOG [13] extends Datalog with a mechanism for dealing with object identities based on Skolem functions. Atomic formulae in ILOG and other Datalog-like languages typically use a positional representation of attributes, making the syntax unsuitable for dealing with relations with lots of attributes. Furthermore, clauses completely describe a target or intensional value in terms of other values. As noted earlier, when targeting complex data-structures it is desirable to have clauses which give partial descriptions of a target value. There query languages are therefore not appropriate in the context of database transformations.

The *F-logic* of Kifer and Lausen [15] is another logic-based language for reasoning about database schemas and instances involving various object-oriented concepts. F-logic differs from *WOL* in blurring the distinctions between objects and schema-classes, and between objects and attributes, and by directly incorporating a notion of inheritance or subsumption. In the design of *WOL* there is a clear distinction between schemas and instances, and between attributes and classes, which were felt to be conceptually distinct. More significantly, *WOL* aims to be a practical language allowing the efficient implementation of a significant class of transformation specifications and constraints, while F-logic does not lend itself to practical implementations.

Possibly the closest existing work to *WOL*, in both spirit and purpose, are the structural manipulations of Abiteboul and Hull [3]. These make use of rewrite rules, which have similar feel to the Horn clauses of *WOL*, but are based on pattern matching against complex data-structures, allowing for arbitrarily nested set, record and variant type constructors. The main contributions of *WOL*, compared with [3], lie in its ability to deal with object-identity and hence recursive data-structures, and in the uniform treatment of transformation rules and constraints.

4 Transformations and Constraints

Since constraints specify valid instances of databases, they can play an important role in implementing as well as in proving the correctness of transformations.

4.1 Determining Transformations using Constraints

One of the most obvious examples of how constraints on a target database play a part in determining a transformations is that of key constraints. In the examples of the previous section, clauses (T1) and (T3) must be combined with a key constraint on $Country_T$ (C3) in order to completely specify an object of class $Country_T$ in the target database.

Other constraints, such as inclusion dependencies and specialization or generalization relations, may also play a part in determining transformations. For example suppose we wish to generalize $Country_T$ and $State_T$ by adding a class $Place_T$, with key attribute *name*, and attributes *currency* and *language*, to our international Cities, Countries and States database. The relationship between objects of class $Place_T$ and those of class $State_T$ or $Country_T$ might be expressed by means of constraints:

$$P \in Place_T,\ P.name = N,\ P.currency = C,\ P.language = L \quad (C6)$$
$$\Leftarrow\ X \in Country_T,\ X.name = N,\ X.currency = C,\ X.language = L;$$

$$P \in Place_T,\ P.name = N,\ P.currency = \text{"US-Dollars"},\ P.language = \text{"English"} \quad (C7)$$
$$\Leftarrow\ S \in State_T,\ X.name = N, X.language = L;$$

These clauses, in combination with key clauses for the class $Place_T$, are sufficient to determine the objects of class $Place_T$, so no additional transformation clauses, other than those already defined for generating the classes $Country_T$ and $State_T$, would be needed.

4.2 Optimizing Transformations using Constraints

Constraints on the source databases do not play a part in determining a transformation, since they only assert restrictions on the source database, which we assume are satisfied prior to the transformation being carried out. Nevertheless they can play an important part in implementing a database transformation.

The implementation of *WOL* described in Section 5 is based on the approach of first deriving new *WOL* clauses from a transformation program which completely describe how to generate an object in the target database, and then applying these new clauses using some underlying database query engine. Source database constraints play an important part in optimizing this process, both by simplifying the derived rules and by causing unsatisfiable rules to be rejected.

Example 4.1: For the schemas of the Cities and Countries databases described earlier, suppose we split the description of the instantiation of the $Country_T$ class over several transformation clauses:

$$X = Mk^{Country_T}(N),\ X.language = L \quad (T4)$$
$$\Leftarrow\ Y \in Country_E,\ Y.name = N,\ Y.language = L$$

$$X = Mk^{Country_T}(N),\ X.currency = C \quad (T5)$$
$$\Leftarrow\ Z \in Country_E,\ Z.name = N,\ Z.currency = C$$

Combining these clauses gives

$$X = Mk^{Country_T}(N),\ X.language = L,\ X.currency = C$$
$$\Leftarrow\ Y \in Country_E,\ Y.name = N,\ Y.language = L$$
$$Z \in Country_E,\ Z.name = N,\ Z.currency = C$$

To apply this clause we would need to take the product of the source class $Country_E$ with itself, and try to bind Y and Z to pairs of objects in $Country_E$ which have the same value on their *name* attribute.

Suppose however, we had a constraint on the source database:

$$X = Y \Leftarrow X \in Country_E\ Y \in Country_E\ X.name = Y.name \quad (C8)$$

That is, *name* is a key for $Country_E$. We could then use this source constraint to simplify our previous, derived transformation clause, in order to form the new clause:

$$X = Mk^{Country_T}(N), X.language = L, X.currency = C$$
$$\Leftarrow\ Y \in Country_E,\ Y.name = N,\ Y.language = L,\ Y.currency = C$$

This clause does not actually give us any new information about the target database, but that it is simpler and more efficient to evaluate. ∎

4.3 Correctness of Transformations

Since many applications of database transformations are highly critical, it is important to have some notion of *correctness* for database transformations, and, in particular, to be able to show that a transformation preserves the *information* stored in a database.

The issues of information capacity and information dominance for database schemas have been thoroughly examined in [11]. The basic concept was defined by saying that one schema *dominates* another iff there is an *injective* transformation, subject to various restrictions, from the instances of the first schema to the instances of the second: that is each distinct source database instance is mapped to a distinct target instance by the transformation. Such a transformation is

said to be *information preserving*. The notion of information capacity and information preserving transformations is complicated by the presence of object identities (see [16]), but can easily be adapted. An important conclusion of [11] is that none of these criteria capture an adequate notion of semantic dominance, that is, whether there is a semantically meaningful interpretation of instances of one schema as instances of another. Consequently the task of ensuring a transformation is semantically meaningful still requires a knowledge and understanding of the databases involved.

In [18, 19] Miller et al. analyze the information requirements that need to be imposed on transformations in various applications. For example they claim that if a transformation is to be used to view and query an entire source database then it must be a total injective function, while if a database is to be updated via a view then the transformation to the view must also be surjective. An important observation in [18] is that database transformations can fail to be information capacity preserving, not because there is anything wrong with the definition of the transformations themselves, but because certain constraints which hold on the source database are not expressed in the source database schema. However the full significance of this observation is not properly appreciated: in fact it is frequently the case that the constraints that must be taken into account in order to validate a transformation have not merely been omitted from the source schema, but are not expressible in any standard constraint language.

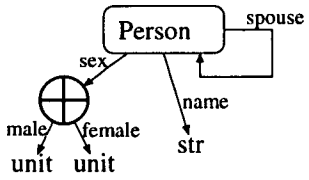

Figure 4: A schema of persons and marriages prior to schema evolution

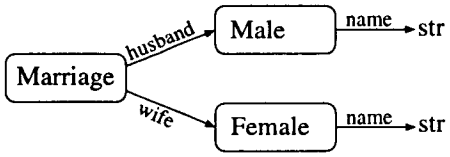

Figure 5: A schema of persons and marriages after schema evolution

Example 4.2: Consider the schema evolution illustrated in Figures 4 and 5. The first schema has only one class, *Person*, with attributes representing a person's *name*, *sex* (a variant of *male* and *female*) and *spouse*. In our second (evolved) schema the *Person* class has been split into two distinct classes, *Male* and *Female*, perhaps because we wished to start storing some different information for men and women. Further the *spouse* attribute is replaced by a new class, *Marriage*, perhaps because we wished to start recording additional information such as dates of marriages, or allow un-married people to be represented in the database.

It seems clear that there is a meaningful transformation from instances of the first database to instances of the second. The transformation can be described by the following *WOL* program:

$$X \in \mathit{Male},\ X.\mathit{name} = N \qquad (T6)$$
$$\Leftarrow\ Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{male}}();$$

$$X \in \mathit{Female},\ X.\mathit{name} = N \qquad (T7)$$
$$\Leftarrow\ Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}();$$

$$M \in \mathit{Marriage},\ M.\mathit{husband} = X,\ M.\mathit{wife} = Y \qquad (T8)$$
$$\Leftarrow\ X \in \mathit{Male},\ Y \in \mathit{Female},\ Z \in \mathit{Person},$$
$$W \in \mathit{Person},\ X.\mathit{name} = Z.\mathit{name},$$
$$Y.\mathit{name} = W.\mathit{name},\ W = Z.\mathit{spouse};$$

Unfortunately this transformation is not information preserving: there are instances of the *spouse* attribute that are allowed by the first schema that will not be reflected by the second schema. In particular the first schema does not require that the *spouse* attribute of a man goes to a woman, or that for each *spouse* attribute in one direction there is a corresponding *spouse* attribute going the other way. However if we augment the first schema with additional constraints, such as:

$$X.\mathit{sex} = \mathit{ins}_{\mathit{male}}() \qquad (C9)$$
$$\Leftarrow\ Y \in \mathit{Person},\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}(),$$
$$X = Y.\mathit{spouse};$$

$$Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}() \qquad (C10)$$
$$\Leftarrow\ X \in \mathit{Person},\ X.\mathit{sex} = \mathit{ins}_{\mathit{male}}(),$$
$$Y = X.\mathit{spouse};$$

$$Y = X.\mathit{spouse} \qquad (C11)$$
$$\Leftarrow\ Y \in \mathit{Person},\ X = Y.\mathit{spouse};$$

we can then show that the transformation is information preserving on those instances of the first schema that satisfy these constraints. These constraints deal with values at the instance level of the database, and could not be expressed with the standard constraint languages associated with most data-models. ■

This highlights a basic problem with information capacity analysis of transformations: such an analysis assumes that schemas give a complete description of the set of possible instances of a database. In practice schemas are seldom

complete, either because certain constraints were forgotten or were not known at the time of schema design, or because the data-model being used simply isn't sufficiently expressive. In the above example, the transformation appears to discard information, while in fact this is because the new schema is a better fit for the data.

WOL takes a step towards addressing these problems: It provides a formalism for expressing constraints necessary in order to ensure information preserving transformations, and allows for formal reasoning about the interactions between database transformations and constraints.

5 Morphase

Implementing a transformation directly using clauses such as (T1), (T2) and (T3) would be inefficient: to infer the structure of a single object we would have to apply multiple clauses, for example clauses (T1), (T3) and (C3) would be needed for a single *Country* object. Further, since some of the transformation clauses, such as (T1) and (T3), involve target classes and objects in their bodies, we would have to apply the clauses *recursively*: having inserted a new object into $Country_T$ we would have to test whether clause (T2) could be applied to that *Country* in order to introduce a new $City_T$ object.

Since *WOL* programs are intended to transform entire databases and may be applied many times, we trade off compile-time expense for run-time efficiency. We pursue an implementation strategy which finds, at compile time, an equivalent, more efficient transformation program in which all clauses are in *normal form*. A transformation clause in normal form completely defines an insert into the target database in terms of the source database only. That is, a normal form clause will contain no target classes in its body, and will completely and unambiguously determine some object of the target database in its head. A transformation program in which all the transformation clauses are in normal form, can easily be implemented in a single pass using some suitable database programming language.

Our prototype implementation of *WOL*, *Morphase*, first rewrites a *WOL* transformation program so that all the clauses are in normal-form, and then uses a database programming language, CPL, to perform the transformations. *Morphase* has been used in several trial transformations within the Philadelphia Genome Center for Chromosome 22 (see [8, 17] for details; also visit the Web site `http://www.cis.upenn.edu/~db/morphase`). A number of syntactic extensions and shorthands were used in order to make programming in *WOL* more convenient and concise without effecting the expressive power of the language.

Unfortunately, not all complete transformation programs have equivalent normal form transformation programs, and it is not decidable whether a transformation program is complete or such an equivalent normal form transformation program exists. Consequently *Morphase* imposes certain syntactic restrictions on transformation programs, to ensure that they are *non-recursive*, such that most natural transformations satisfy these restrictions.

The architecture of *Morphase* is illustrated in figure 6: *WOL* transformation rules are typically written by the user of the system; however a large number of constraints, such as keys and other dependencies, can be automatically generated from the meta-data associated with the source and target databases, in order to complete a transformation program. Such constraints are time consuming and tedious to program by hand. Deriving them directly from meta-data therefore reduces the amount of grunge work that needs to be done by the programmer, and allows him or her to concentrate on the structural part of a transformation.

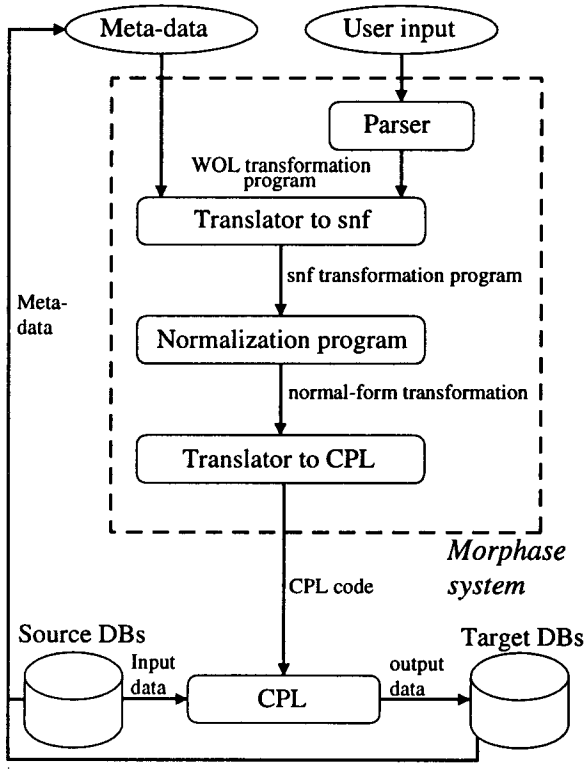

Figure 6: Architecture of the *Morphase* system

The translation of a *WOL* transformation program has several stages. The clauses are first rewritten into *semi-normal form* (*snf*), which reduces the number of forms the atoms of a clause can take, so that any two equivalent clauses or sets of atoms will differ only in their choice of variables. This simplifies the unification of clauses, as well as the book-

keeping necessary for optimizations. The snf program is then transformed into a normal-form program if possible, using a process of unifying and unfolding clauses. A number of important optimizations are used during this process, including the application of source and target constraints to simplify clauses. See [17] for details.

Note that this *normalization* process differs from the unification and unfolding of clauses carried out in certain implementations of Horn-clause logic based languages such as Prolog and Datalog, in that it deals with un-instantiated clauses for the purpose of forming new clauses or *re-writing* a transformation program. In traditional logic based languages, unfolding and unification are used in order to apply clauses and to bind the variables of a query to actual values.

Once translated into normal-form, a *WOL* program can be executed against the source databases to produce the target database. Complete, normal-form *WOL* programs are compiled into CPL, a database programming language for complex values developed at the University of Pennsylvania. The reason for using CPL as an implementation language is that it supports many of the data-types that we are interested in, and its implementation in the Kleisli system [7] enables us to access a wide variety of heterogeneous database systems, including those that we wished to use in our trials. We can currently connect to Sybase, Oracle, ASN.1 and ACeDB[2] databases, as well as a number of application programs such as the sequence analysis packages Blast and FASTA; extending Kleisli to incorporate new databases is relatively easy. However, translating normal form *WOL* programs into some other sufficiently expressive DBPL should be a straightforward task.

6 Conclusions

The development of *WOL* was in part motivated by experiences in informatics support of the Philadelphia Human Genome Center for Chromosome 22. As is the case in many biomedical databases, the database for sequence data pertaining to Chromosome 22 at Penn (Chr22DB) undergoes frequent schema evolution due to changing experimental techniques, resulting in the need for various data transformations to maintain data input screens and analysis packages. Additional data for Chr22DB (a Sybase database) is imported from ACe22DB, an ACeDB database located at the Sanger Centre in Cambridge, England. ACeDB represents data in tree-like structures with object identities, and is well suited for representing "sparsely populated" data [23]. Data is structured very differently in ACe22DB and Chr22DB, since they use incompatible data-models as well as different interpretations of the underlying data and how it should be structured. Therefore, complicated programs were hand-coded to periodically exchange data between ACe22DB and Chr22DB. However, both of the databases are constantly evolving necessitating modifications of the hand-coded programs.

WOL and its implementation in *Morphase* have been tested on these and other transformations arising at the Bio-Informatics center at Penn, and the performance of the system has been evaluated in terms of ease of use, compilation time, and size and complexity of the resulting normal form program. The size and complexity of the normal form program is an indirect indicator of the execution time of the actual transformation, since the Kleisli optimizer will rewrite the CPL code to a more efficient form. These trials gave us significant insights into optimizations for the normalization procedure and the translation to CPL, many of which were incorporated into the prototype.

WOL transformations were programmed by researchers connected with the Philadelphia Genome Center, who used their domain knowledge to formulate the transformations. Learning *WOL* and using it to express the transformations was found to be an easy and natural process. The most difficult part of the process was in fact understanding the foreign databases. As ACe22DB and Chr22DB have evolved over time, it has also been easy to modify the original *WOL* program to reflect schema changes.

The time taken to compile and normalize various *WOL* programs was also measured. These included programs in which all the clauses were already in normal form, which represented the minimum time necessary for the system to process a transformation program. It was found that the use of constraints was extremely important in gaining acceptable performance from the implementation, with a non-normalized transformation program with constraints taking approximately six times longer to compile than a normalized program. If constraints were omitted the time taken to normalize a program, and the size of the resulting normal-form program, could be exponential in the size of the original program.

The *WOL* language has also been used independently by researchers in the VODAK project at Darmstadt, Germany, [1], in order to build a data-warehouse of protein and protein-ligand data for use in drug design. This project involved transforming data from a variety of public molecular biology databases, including SWISSPROT and PDB, and storing it in an object-oriented database, ReLiBase. *WOL* was used to specify structural transformations of data, and to guide the implementations of these transformations.

The primary contributions of this paper lie in the desiderata we have gleaned for database transformation languages, and the language *WOL* itself. While the idea of using a Horn-

[2] A c. elegans Database (AceDB). While "C elegans" may sound like a contradiction in terms to a computer scientist, it is actually a small worm.

clause logic query language for complex object databases is not new, several features of *WOL* make it uniquely applicable as a database transformation language. First, *WOL* is designed to deal with transformations involving the complex and possibly recursive data-structures that occur in object-oriented databases, as well as with traditional relational databases involving large numbers of attributes. This necessitates the ability to specify partial rules, which in turn requires some unique, efficient compile time rewriting strategies. Second, the ability to capture and reason about database constraints in the same formalism allows for the use of constraints in determining and optimizing transformations, and in reasoning about the correctness of transformations. We have also briefly described implementation strategies, based on the *WOL* language, which prototype implementations and trials have shown to be extremely promising.

In future work, we also wish to explore how well the approach scales to larger problems than we have considered. (Human Genome Project databases are typically rather small, less than a gigabyte.) It is also clear that there is a potential for graphical schema manipulation tools generating *WOL* transformation programs. Finally, given that constraints are available to reason about formally within the *WOL* framework, we would like to explore ways in which to reason about the "correctness" of transformations.

Acknowledgments. We would like to thank Peter Buneman for his help,suggestions and examples, Barbara Eckman and Carmem Hara for their help with the *Morphase* trials, and Scott Harker for his implementation work on *Morphase*.

References

[1] K. Aberer and K. Hemm. A methodology for building a data warehouse in a scientific environment. In *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS), Brussels, Belgium*, June 1996.

[2] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[3] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[5] F. Bancilhon. Object-oriented database systems. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 152–162, Los Angeles, California, 1988.

[6] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, 1987.

[7] P. Buneman, S. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proceedings of 21st VLDB*, September 1995. Also Technical report MS-CIS-95-10, Dept. of Computer and Information Science, University of Pennsylvania. March 1995.

[8] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. Technical Report MS-CIS-93-94/L&C 74, University of Pennsylvania, Philadelphia, PA 19104, December 1994.

[9] U. Dayal and H. Hwang. View definition and generalisation for database integration in Multibase: A system for heterogeneous distributed databases. *IEEE Transactions on Software Engineering*, SE–10(6):628–644, November 1984.

[10] N. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[11] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):865–886, August 1986.

[12] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[13] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[14] S. N. Khoshafian and G. P. Copeland. Object identity. In S. B. Zdonik and D. Maier, editors, *Readings in Object Oriented Database Systems*, pages 37–46. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[15] M. Kifer and G. Laussen. F-logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proceedings of ACM-SIGMOD 1989*, pages 46–57, June 1989.

[16] A. Kosky. Observational properties of databases with object identity. Technical Report MS-CIS-95-20, Dept. of Computer and Information Science, University of Pennsylvania, 1995.

[17] A. Kosky. *Transforming Databases with Recursive Data Structures*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, November 1995.

[18] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th International VLDB Conference*, pages 120–133, August 1993.

[19] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19, 1994.

[20] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Transactions on Software Engineering*, SE-13(7):785–798, July 1987.

[21] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of data models as canonical models for federated databases. *SIGMOD Record*, 20(4):44–48, December 1991.

[22] P. Shoval and S. Zohn. Binary-relationship integration methodology. *Data and Knowledge Engineering*, 6:225–249, 1991.

[23] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.